データベース第11回

第12章　障害時回復

障害の種類

- トランザクション障害
  - 誤った入力の読込，所望のデータが見つからない，計算途中のオーバフロー，必要な資源不足，などによる異常終了
- システム障害
  - DBMSやOSの障害によるシステム暴走やフリーズ，ハードウェアのバグや電源断など．揮発メモリ上のデータは失われるが，不揮発性の2次記憶中のデータは生き残っている
- メディア障害
  - 2次記憶装置の障害で，ディスククラッシュなど

主記憶と2次記憶

- 仮想記憶システム
  - 永続的なデータは，2次記憶中に存在
  - 計算機で処理するためには，データは主記憶中になければならない
- 障害に対して，いろいろなケースがある
  - 障害発生時点でトランザクション実行中
  - 障害発生時点でコミット完了
    - 主記憶上の変更はすべて2次記憶に反映済み
    - 主記憶上の変更の一部は主記憶上にのみあり，2次記憶には反映されていない

ログ法

- ログ
  - データベースに対する処理の記録
  - 時系列データファイル
  - 障害時回復に必要な情報なので，障害の発生しない不揮発性メディア（例えば，多重化されたディスクシステム）に保存される

揮発・不揮発：揮発は，電源が切れるとなくなる．不揮発は，それにかかわらず生き残る性質をいう．

T1とT2のログの例(図12.2)

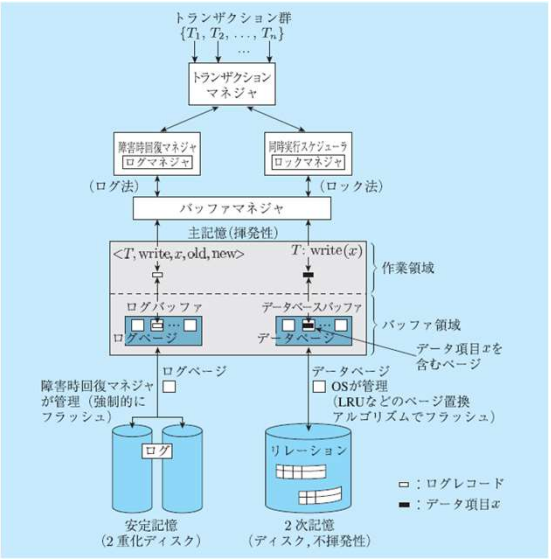
T_1	T_2
begin	begin
read(x)	read(y)
write(x)	write(y)
end	end

(a) トランザクション T_1とT_2

ログ

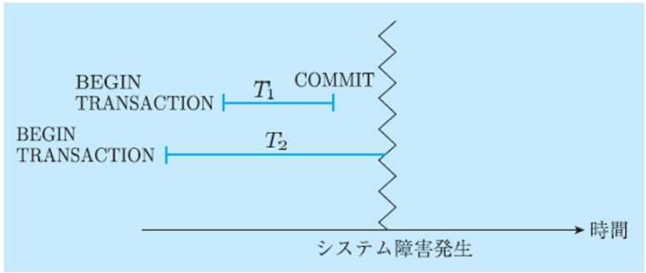
LogSN	TransID	作用	RelID	TupleID	ColID	旧値	新値
⋮							
1054	T_1	BEGIN					
1055	T_1	READ	社員	007	給与		
1056	T_2	BEGIN					
1057	T_1	WRITE	社員	007	給与	50	100
1058	T_2	READ	社員	002	給与		
1059	T_1	COMMIT					

(b) T_1とT_2の同時実行を記録しているログ



WALプロトコル

- WAL: write ahead log
  - まずログをとる，というプロトコル
- 2次記憶への書込みが，データベース内のデータの変更を反映させるものと，ログをとるものの2種類ある
- ログファイルへの書込みは強制書込み
  - ディスクへのフラッシュを確実に行う
- データベース内のデータの変更の書込みのタイミング
  - 書込むタイミングを全くOSに任せるのが即時更新，終了ログが書込まれるのを待つのが遅延更新
  - 遅延更新では，変更のログとCOMMITログが全てディスクに書き込まれた後にならないと，データの更新は一切反映されない



Undo/Redo障害時回復

- ルール：ディスク上のデータに対する変更の前に，それについてのログを書込む(即時更新)
- このときの障害時回復法
  1. commitしたすべてのトランザクションを，earliest-firstの順番でRedoする
  2. 完全に終わらなかったすべてのトランザクションを，latest-firstの順番でUndoする

9

No-Undo/Redo障害時回復

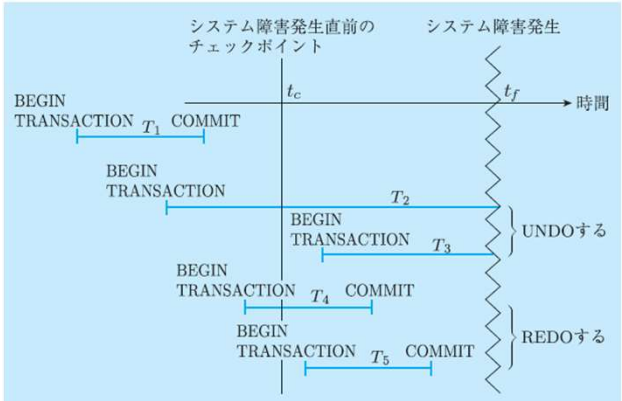
- ルール：ディスク上のデータの変更は，そのデータに対するcommitまでのすべてのログが取られた後に行う(遅延更新)
- このときの障害時回復法
  1. commitしたトランザクションを調べる
  2. ログを先頭からスキャンする．データの変更のログが見つかった場合，
     a. そのトランザクションがcommitしてなければ，なにもしない
     b. そのトランザクションがcommitしていたら，Redoする
  3. 完全に終わらなかった各トランザクションについては，ABORTのログを作って，ログファイルに書込む

10

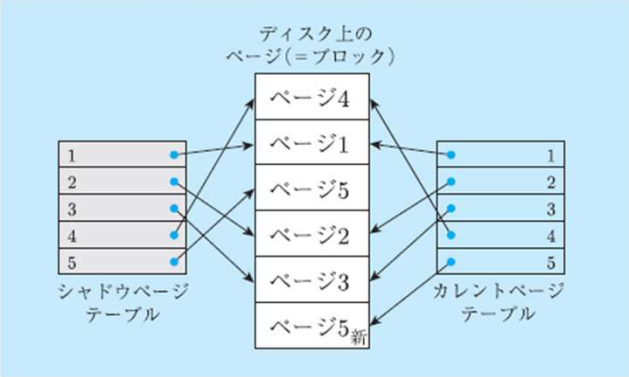
チェックポイント

- 適切な時点で，ログとデータの更新の両方をディスクに書込んでおけば，障害時回復のコストを少なくできる
- この「書込み」を強制的に行うのが，チェックポイント
- チェックポイントで行われる処理
  1. 実行中のトランザクションをすべて一時停止
  2. データベースバッファの内容をデータベースに強制書出し
  3. チェックポイントレコードをログに書き出す
  4. 中断させていたトランザクションを再開

11

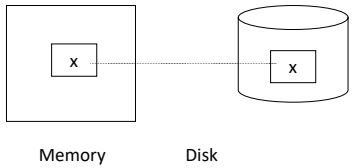


12



Second order of business:

Storage hierarchy



Key problem Unfinished transaction

Example Constraint: A=B

T_1: A ← A × 2

B ← B × 2

Redo-Loggingにおける前提

- ディスク上にあるデータベース内のデータXを更新する前に，Xに対する操作とそのトランザクションの終了のログをディスク上に書き込む
- Xに対する操作とそのトランザクションの終了のログ：<T, X, v> と <COMMIT T>
- ディスク上のXの更新が，<COMMIT T>までのすべてのログをディスク上に保存するまで遅らされるので，これを遅延更新という

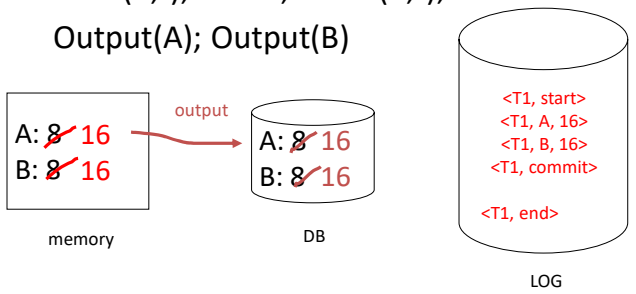
Redo-Loggingでの動作とそのログの例

Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t:=t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<T, A, 16>
5)	READ(B,t)	8	16	8	8	8	
6)	t:=t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<T, B, 16>
8)							<COMMIT T>
9)	FLUSH LOG						
10)	OUTPUT(A)	16	16	16	16	8	
11)	OUTPUT(B)	16	16	16	16	16	

17

Redo logging (deferred modification)

T1: Read(A,t); t←t×2; write (A,t);
Read(B,t); t←t×2; write (B,t);
Output(A); Output(B)



18

Redo障害時回復

- ルール：ディスク上のデータの変更は，そのデータに対するcommitまでのすべてのログが取られた後に行う(遅延更新)
- このときの障害時回復法
  1. commitしたトランザクションを調べる
  2. ログを先頭からスキャンする．データの変更のログが見つかった場合，
     a. そのトランザクションがcommitしてなければ，なにもしない
     b. そのトランザクションがcommitしていたら，Redoする
  3. 完全に終わらなかった各トランザクションについては，ABORTのログを作って，ログファイルに書込む

19

Key drawbacks:

- *Undo logging:* cannot bring backup DB copies up to date
- *Redo logging:* need to keep all modified blocks in memory until commit

20

Solution: undo/redo logging!

Update ⇒ <Ti, Xid, Old X val, New X val>
page X

Undo/Redo-Logging の前提

- あるトランザクションTによるディスク上のデータベース内のデータXへの更新を行う前に，その更新のログを必ずディスク上に書き込む
- Xに対する更新のログ：<T, X, v, w>
- ディスク上のXの更新は，ログの後であればいつでもよいので，即時更新という

Undo/Redo-Loggingでの動作とそのログの例

Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t:=t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<T, A, 8, 16>
5)	READ(B,t)	8	16	8	8	8	
6)	t:=t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<T, B, 8, 16>
8)	FLUSH LOG						
9)	OUTPUT(A)	16	16	16	16	8	
10)							<COMMIT T>
11)	OUTPUT(B)	16	16	16	16	16	

Undo/Redo障害時回復

- ルール：ディスク上のデータに対する変更の前に，それについてのログを書込む(即時更新)
- このときの障害時回復法
  1. commitしたすべてのトランザクションを，earliest-firstの順番でRedoする
  2. 完全に終わらなかったすべてのトランザクションを，latest-firstの順番でUndoする

第13章　同時実行制御
ー同時実行制御とはー

25

トランザクションの同時実行

- n個のトランザクション群{T1,T2,・・・,Tn}の実行処理リクエストが発生
- n個のトランザクションを一個一個順番に処理する=直列実行
  - 一貫性は保たれる（なにも問題は発生しない）
  - しかし，処理効率は良くない
- 効率(単位時間当たりのトランザクションの処理数)向上のため，同時実行を考える
  - OSのマルチプログラミング・マルチタスクと同様
  - 同時実行=並行実行≠並列実行

26

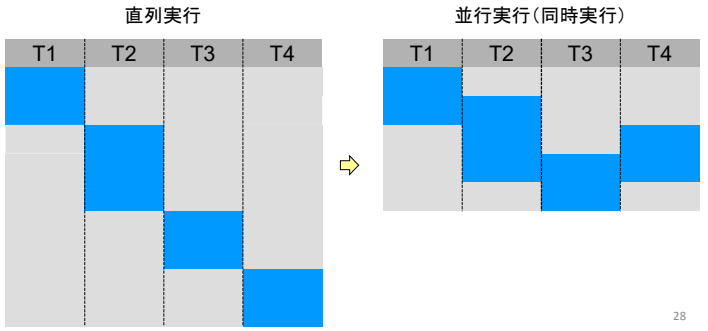
トランザクションの同時実行

- 実際のデータベースでは，複数のトランザクションを並列に実行する
  - トランザクションの途中でディスク読み書きなどが生じると，その入出力を待っている間，CPUの処理は停止する
  - トランザクションが並行に実行していれば，あるトランザクションが入出力待ちで停止している間も，別のトランザクションを実行できるので効率的
  - さらに，マルチCPU，マルチコアのコンピュータであれば，常に複数のトランザクションを同時に実行可能（＝並列実行）

27

トランザクションの同時実行

- 結果が同じであれば，なるべく並列に実行した方が効率的



28

同時実行時の注意点

- あるトランザクションが，他のトランザクションに影響を与えないようにする必要がある
- 複数のトランザクションが同時に同じデータを読み書きしようとすると不具合が生じることがある

単純なトランザクションのステップ

```
begin
   read(x)
   write(x)
end
```

読み(read)書き(write)の系列と考える（図13.1）

2個のトランザクションの同時実行スケジュール

```
T1                    T2
begin                 begin
   read(x)               read(x)
   write(x)              write(x)
end                      write(y)
                      end
```

T1	T2
read(x)	--
--	read(x)
write(x)	--
--	write(x)
--	write(y)

T1	T2
read(x)	--
--	read(x)
--	write(x)
write(x)	--
--	write(y)

T1	T2
read(x)	--
--	read(x)
--	write(x)
--	write(y)
write(x)	--

T1	T2
read(x)	--
write(x)	--
--	read(x)
--	write(x)
--	write(y)

同時実行制御の必要性

- 同時実行を正しく制御しないと，異状が発生する可能性がある
- 異状にはさまざまある
  - 遺失更新異状 (lost update anomaly)
  - 不整合検索異状 (inconsistent retrieval anomaly)
  - 汚読異状 (dirty read anomaly)

遺失更新異状

T1の実行	T2の実行
read(A)	--
--	read(A)
write(A:=A-30)	--
--	write(A:=A-20)
COMMIT	--
--	COMMIT

33

汚読異状

T1の実行	T2の実行
read(A)	--
write(A:=A+10)	--
--	read(A)
--	write(A:=A-10)
--	COMMIT
ROLLBACK	--

34

反復不可能な読み異状

T1の実行	T2の実行
read(A)	--
--	read(A)
--	write(A:=A-1)
--	COMMIT
read(A)	--

35

直列化可能性(serializability)

- 「異状のない同時実行」のための概念
- n個のトランザクションを同時実行する場合，ある直列スケジュールが存在して，両者の実行結果が同じであれば，その同時実行スケジュールは直列化可能といい，異状のない一貫性のある結果が得られる.

36

二つのトランザクションの場合

T1の実行	T2の実行
read(x)	--
--	read(y)
write(x)	--
--	read(x)
--	write(y)

T1の実行	T2の実行
read(x)	--
write(x)	--
--	read(y)
--	read(x)
--	write(y)

T1の実行	T2の実行
--	read(y)
--	read(x)
--	write(y)
read(x)	--
write(x)	--

これは直列化可能？

T1	T2	T3	T4	T5
--	--	--	--	read(x)
--	read(x)	--	--	--
--	--	--	--	write(x)
read(x)	--	--	--	--
--	--	--	read(x)	--
--	read(y)	--	--	--
write(x)	--	--	--	--
--	--	read(x)	--	--
--	write(y)	--	--	--
--	--	--	read(y)	--
--	--	write(x)	--	--
--	--	--	write(y)	--

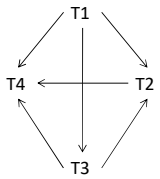
第14章　同時実行制御
ースケジュール法とロック法ー

スケジュールの相反直列化可能性

- トランザクション群のスケジュールSが与えられたとき，Sに従って実行して正しい結果が得られるかを検証する
- 相反グラフ解析 (conflict graph analysis)

スケジュール S

T1	T2	T3	T4
--	--	read(x)	--
read(x)	--	--	--
--	--	write(x)	--
--	read(x)	--	--
read(y)	--	--	--
--	--	--	read(x)
write(y)	--	--	--
--	read(y)	--	--
--	--	--	write(x)
--	write(y)	--	--



相反グラフ

Sの相反グラフCG(S)

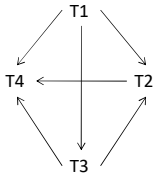
1. CG(S)はn個のノードT1, T2,・・・,Tnからなる
2. ノードTiからTjに有向辺が張られるのは，あるデータ項目xが存在し，S中で次のいずれかが成立するとき
   1. ステップTi:read(x)がステップTj:write(x)に先行する
   2. ステップTi:write(x)がステップTj:write(x)に先行する
   3. ステップTi:write(x)がステップTj:read(x)に先行する

相反直列化可能性

- スケジュールSが相反直列化可能とは，ある直列スケジュールS'が存在して，SとS'が相反等価のときをいう．ここに，一般にスケジュールSとS'が相反等価とは，相反しているすべての2つのステップの順番が，SとS'で同一のときをいう．
- スケジュールSが相反直列化可能であるための必要かつ十分条件はSの相反グラフCG(S)が非巡回であること．また，Sに相反等価な直列スケジュールはCG(S)をトポロジカルソートすることにより得られる．

トポロジカルソート

- 有限グラフCG(S)にサイクルがなければ，入ってくる有向辺がないノードが少なくとも一つ存在．これをT(1)とする．CG(S)からT(1)とそれに結合している有向辺を取り除き，同様な処理を繰り返してトランザクションの系列を得る



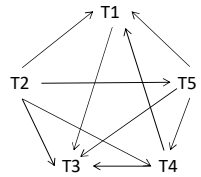
T1, T3, T2, T4

相反直列化可能性検証

1. スケジュールSの相反グラフCG(S)作成
2. CG(S)が非巡回かどうか検証
3. 非巡回ならSは相反直列化可能．そうでないなら，相反直列化可能ではない

これは直列化可能？

T1	T2	T3	T4	T5
--	--	--	--	read(x)
--	read(x)	--	--	--
--	--	--	--	write(x)
read(x)	--	--	--	--
--	--	--	read(x)	--
--	read(y)	--	--	--
write(x)	--	--	--	--
--	--	read(x)	--	--
--	write(y)	--	--	--
--	--	--	read(y)	--
--	--	write(x)	--	--
--	--	--	write(y)	--



T2,T5,T4,T1,T3

静的・動的スケジュール

- 相反直列化可能を検証した先のスケジュール法は，静的スケジュール法．つまり，Sが与えられたときに，それが直列化可能かどうかを調べる．
  - バッチ処理などのときに使われる
- トランザクション処理要求がシステムに動的に入ってくる状況でのスケジュール法が別に必要．ここではロック法を解説

ロック法

- ロッキングプロトコル

1. トランザクションは，データ項目xを読むにしろ書くにしろ，それを行う前にまずxをロックしなければならない
2. もしロックしようとしたデータ項目がほかのトランザクションによりロックされているなら，それをロックすることはできない
3. トランザクションは，データ項目のロックが不必要となったら，アンロックする

	lock	-
lock	偽	真
-	真	真

ロック法詳細

- 単にロック法のプロトコルに従っただけでは，相反直列化可能性を保証しない

T1の実行	T2の実行
read(A)	--
--	read(A)
write(A:=A-30)	--
--	write(A:=A-20)
COMMIT	--
--	COMMIT

実行
T1:lock(A)
T1:read(A)
T1:unlock(A)
T2:lock(A)
T2:read(A)
T2:unlock(A)
T1:lock(A)
T1:write(A)
T1:unlock(A)
T2:lock(A)
T2:write(A)
T2:unlock(A)

2相ロッキングプロトコル

- 2 phase locking protocol (2PL)

1. トランザクションは，データ項目xを読むにしろ書くにしろ，それを行う前にまずxをロックしなければならない．
2. もしロックしようとしたデータ項目がほかのトランザクションによりロックされているならば，それをロックすることはできない．
3. トランザクションは，データ項目のロックが不必要となったら，アンロックする．しかし，トランザクションは，読み書きのために必要なすべてのロックが完了する以前に，それらをアンロックすることはしない．

- 2PLに従えば，相反直列化可能なスケジュールになる

2PLの例

```
begin
    read(x)
    read(y)
    write(x)
    write(y)
end
```

(a) トランザクション T

時刻	T	
t_1	lock(x)	第 1 相（成長相）
t_2	read(x)	
t_3	lock(y)	
t_4	read(y)	
t_5	write(x)	
t_6	unlock(x)	第 2 相（縮退相）
t_7	write(y)	
t_8	unlock(y)	

(b) 2PL に従ったトランザクション T の実行


```
begin
    read(x)
    write(x)
end
```

(a) トランザクション T_1

```
begin
    read(y)
    read(x)
    write(y)
end
```

(b) トランザクション T_2

時刻	T_1	T_2
t_1	lock(x)	—
t_2	read(x)	—
t_3	—	lock(y)
t_4	—	read(y)
t_5	write(x)	—
t_6	unlock(x)	—
t_7	—	lock(x)
t_8	—	read(x)
t_9	—	write(y)
t_{10}	—	unlock(x)
t_{11}	—	unlock(y)

(c) 2PL に従うトランザクション T_1 と T_2 の同時実行の順法スケジュール

時刻	T_1	T_2
t_1'	lock(x)	—
t_2'	read(x)	—
t_3'	write(x)	—
t_4'	unlock(x)	—
t_5'	—	lock(y)
t_6'	—	read(y)
t_7'	—	lock(x)
t_8'	—	read(x)
t_9'	—	write(y)
t_{10}'	—	unlock(x)
t_{11}'	—	unlock(y)

(d) (c) に等価な直列スケジュール

厳格な2PL

- 2PLに従えば，相反直列化可能スケジュールになる
- しかし，第1相(成長相，ロック相)と第2相(縮退相，アンロック相)の境目を適切に設定するのは難しい
- 厳格な2PL
  - コミットかアボートに到達したら，COMMITやROLLBACKの直前にすべてのロックをアンロックする．それまではアンロックは一切しない
  - ただし，同時実行度は低下する

占有ロックと共有ロック

要求＼状態	e-lock	s-lock	−
e-lock	偽	偽	真
s-lock	偽	真	真
−	真	真	真

デッドロック

T_1	T_2
begin	begin
read(x)	read(y)
read(y)	read(x)
write(x)	write(y)
end	end

(a) トランザクション T_1 とトランザクション T_2

時刻	T_1	T_2
t_1	lock(x)	—
t_2	—	lock(y)
t_3	read(x)	—
t_4	—	read(y)
これ以降はロック両立性行列に順法であろうとする限り進めない		

(b) デッドロックの発生

デッドロック

- 待ちグラフ(wait-for graph)を作成し，ループが発見されたら，デッドロックが生じていることがわかる
- この場合，どれかをアボートする

- 待ちグラフを作成するタイミングが重要
  - 待ちグラフをどのくらいの間隔で作るかは，効率とデッドロック検出頻度のトレードオフで難しい問題

デッドロック

- タイムアウトを行う方法
  - 一定時間トランザクションが進行しなかったら，デッドロックが発生している可能性があるとする
  - 正確な検出は行わない
  - タイムアウト時間の設定が重要

デッドロック

- 2PLにおいて，第1相で必要なロックを前もってすべて取得してから実行を開始する
  - ロックが成功したら，トランザクションの途中でロックが必要になることはないので，デッドロックが原因でトランザクションが失敗することはなくなる
  - 同時実行性は悪い

講義スタイル

- スライド（および板書？）
  - ノートを取る時間を考慮して進めます
  - スライドの進め方が早い時は，手を挙げるなどして止めて下さい
  - ノート取りたかったのに取れなかったところは，聞きに来てください
- 成績
  - 出席：2/3以上なければ，受験資格なし
  - 期末試験

定期試験

- 範囲は，講義したすべての内容
- 1枚(A4サイズ)の片面手書きのメモの持ち込みを認めます
  - テストとメモの筆跡は合わせること
  - 学生番号・名前を必ず書くこと
  - 持ち込んだメモは，必ず解答と一緒に提出してください．採点することがあります